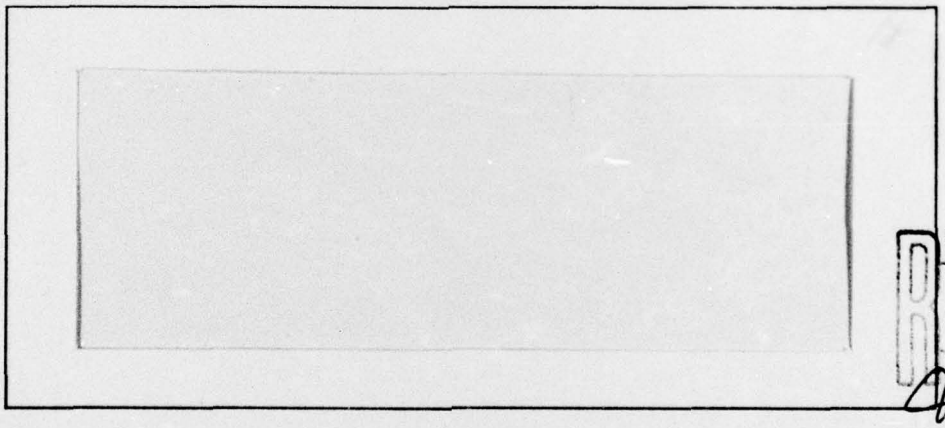
AD-A033 486 YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE F/G 9/2
CONSIDERATIONS IN THE DESIGN OF SOFTWARE FOR SPARSE GAUSSIAN EL--ETC(U)
UNCLASSIFIED 1975 S C EISENSTAT, M H SCHUTZ, A H SHERMAN N00014-67-A-0097-0016
RR-55 NL

1 OF 1
AD
A033486

END

DATE
FILMED
2-77



YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

(12)

(11) 1975

(12) 11 p.

(9) Research (Rept.)

(6) Considerations in the Design of Software for Sparse Gaussian Elimination.

(10) S. C. Eisenstat, M. H. Schultz and A. H. Sherman

Research Report #55

(14) RR-55

This research was supported in part by the Office of Naval Research, N0014-67-A-0097-0016.

(15) N00014-67-A-0097-0016

407051

B

1. Introduction

Consider the large sparse system of linear equations

$$Ax = b \tag{1.1}$$

where A is an N×N sparse nonsymmetric matrix and x and b are vectors of length N. Assume that A can be factored in the form

$$A = LDU \tag{1.2}$$

where L is a unit lower triangular matrix, D a nonsingular diagonal matrix, and U a unit upper triangular matrix. Then an important method for solving (1.1) is sparse Gaussian elimination, or, equivalently, first factoring A as in (1.2) and then successively solving the systems

$$Ly = b,\ Dz = y,\ Ux = z. \tag{1.3}$$

Recently, several implementations of sparse Gaussian elimination have been developed to solve systems like (1.1) (cf. Curtis and Reid [2], Eisenstat, Schultz, and Sherman [5], Gustavson [6], and Rheinboldt and Mesztenyi [7]). The basic idea of all of these is to factor A and compute x without storing or operating on zeroes in A, L, or U. Doing this requires a certain amount of storage and operational overhead, i.e. extra storage for pointers in addition to that needed for nonzeroes and extra nonnumeric "bookkeeping" operations in addition to the required arithmetic operations. All these implementations of sparse Gaussian elimination generate the same factorization of A and avoid storing and operating on zeroes. Thus, they all have the same costs as measured in terms of the number of nonzeroes in L and U or the number of arithmetic operations performed. The implementations do, however, have different overhead requirements, and thus their total storage and time requirements vary a great deal.

In this paper we discuss the design of sparse Gaussian elimination codes. We are particularly interested in the effects of certain flexibility and cost constraints on the design, and we examine possible tradeoffs among the design goals of flexibility, speed, and small size.

In Section 2 we describe a basic design due to Chang [1], which has been used effectively in the implementations referred to above. Next, in Section 3 we discuss the

storage of sparse matrices and present two storage schemes for use with sparse Gaussian elimination. In Section 4 we describe three specific implementations that illustrate the range of possible tradeoffs among design goals. Finally, in Section 5 we give some quantitative comparisons of the three implementations.

2. A Basic Implementation Design

Several years ago Chang [1] suggested a design for the implementation of sparse Gaussian elimination that has proved to be particularly robust. He proposed breaking the computation up into three distinct steps: symbolic factorization (SYMFAC), numeric factorization (NUMFAC), and forward- and back-solution (SOLVE). The SYMFAC step computes the zero structures of L and U (i.e. the positions of the nonzeroes in L and U) from that of A, disregarding the actual numerical entries of A. The NUMFAC step then uses the structure information generated by SYMFAC to compute the numerical entries of L, D, and U. Finally, the SOLVE step uses the numerical factorization generated by NUMFAC to solve the system (1.3).

The main advantage of splitting up the computation as described here is flexibility. If several linear systems have identical coefficient matrices but different righthand sides, then only one SYMFAC and one NUMFAC are needed; the different righthand sides require only separate SOLVE steps. (This situation arises in the use of the chord method to solve a system of nonlinear equations.) Similarly, a sequence of linear systems, all of whose coefficient matrices have identical zero structure but different numerical entries, can be solved using just one SYMFAC combined with separate NUMFAC and SOLVE steps for each system. (This situation arises when Newton's method is used to solve a system of nonlinear equations.)

A drawback to the three-step design is that it is necessary to store the descriptions of the zero structures and the actual numerical entries of both L and U. In essence, the great flexibility is paid for with a large amount of extra storage. By giving up some flexibility, it is possible to reduce substantially the storage requirements. For example, combining NUMFAC and SOLVE into a single NUMSLV step eliminates the need for storing the numerical entries of L.

It is no longer possible, however, to handle multiple righthand sides so efficiently. Then again, if all three steps are combined into one TRKSLV step, it is unnecessary to store even a description of the zero structure of L. But by combining steps in this way, we lose the ability to solve efficiently sequences of systems all of whose coefficient matrices have identical zero structure.

3. Storage of Sparse Matrices

In this section we describe two storage schemes that can be used to store A, L, and U. The schemes are designed specifically for use with sparse Gaussian elimination and they exploit the fact that random access of sparse matrices is not required.

We call the first storage scheme the uncompressed storage scheme. It has been used previously in various forms by Gustavson [6] and Curtis and Reid [2]. The version given here is a row-oriented scheme in which nonzero matrix entries are stored row by row, although a column-oriented version would work as well. Within each row, nonzero entries are stored in order of increasing column index. To identify the entries of any row, it is necessary to know where the row starts, how many nonzero entries it contains, and in what columns the nonzero entries lie. This extra information describes the zero structure of the matrix and is the storage overhead mentioned earlier.

Storing the matrix A with the uncompressed scheme requires three arrays (IA, JA, and A), as shown in Figure 3.1. The array A contains the nonzero entries of A stored row by row. IA contains N+1 pointers that delimit the rows of nonzeroes in the array A -- A(IA(I)) is the first stored entry of the I*th* row. Since the rows are stored consecutively, the number of entries stored for the I*th* row is given by IA(I+1) - IA(I). (IA(I+1) is defined so that this holds for the N*th* row.) The array JA contains the column indices that correspond to the nonzero entries in the array A -- if A(K) contains a_{IJ}, then JA(K) = J. The storage overhead incurred by using the uncompressed storage scheme for A is the storage for IA and JA. Since IA has N+1 entries and JA has one entry per entry of the array A, the storage overhead is approximately equal to

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & 0 \\ 0 & a_{32} & a_{33} & 0 & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & a_{46} \\ 0 & a_{52} & 0 & 0 & a_{55} & 0 \\ 0 & 0 & 0 & a_{64} & 0 & a_{66} \end{bmatrix}$$

A:	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{32}	a_{33}	a_{42}	a_{44}	a_{46}	a_{52}	a_{55}	a_{64}	a_{66}
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
JA:	1	2	1	2	3	4	5	2	3	2	4	6	2	5	4	6
IA:	1	3	8	10	13	15	17									

Figure 3.1.

---

the number of nonzeroes in the matrix A.

Previous implementations of sparse Gaussian elimination have also used variants of the uncompressed storage scheme for storing L and U, as shown in Figure 3.2. Again the storage overhead is approximately equal to the number of nonzero entries in the two matrices. Storing L and U in this way has the advantage that the operational overhead in implementation is quite small, since the data structures are simple and the matrix entries can be accessed quickly.

In certain situations where storage is at a premium, however, it is essential to reduce the storage overhead, even if the operational overhead is increased. This can be done by storing L and U with a more complex compressed storage scheme (cf. Eisenstat, Schultz, and Sherman [4], Sherman [8]). The compressed storage scheme will incur more operational overhead than the uncompressed scheme but the storage requirement will be substantially reduced. In the compressed storage scheme L is stored by columns (or, equivalently, L^T is stored by rows) and U is stored by rows. Figure 3.3 illustrates the derivation of the compressed storage scheme for U. (The derivation for L is similar, using a column-oriented scheme.)

Figure 3.3a shows the data structures required to store U in the uncompressed storage scheme. It is immediately evident that the diagonal entries do not need to be stored, since

$$L = \begin{bmatrix} 1 & & & & & \\ \ell_{21} & 1 & & & & \\ 0 & \ell_{32} & 1 & & & \\ 0 & \ell_{42} & \ell_{43} & 1 & & \\ 0 & \ell_{52} & \ell_{53} & \ell_{54} & 1 & \\ 0 & 0 & 0 & \ell_{64} & \ell_{65} & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & u_{12} & 0 & 0 & 0 & 0 \\ & 1 & u_{23} & u_{24} & u_{25} & 0 \\ & & 1 & u_{34} & u_{35} & 0 \\ & & & 1 & u_{45} & u_{46} \\ & & & & 1 & u_{56} \\ & & & & & 1 \end{bmatrix}$$

L:	1	ℓ_{21}	1	ℓ_{32}	1	ℓ_{42}	ℓ_{43}	1	ℓ_{52}	ℓ_{53}	ℓ_{54}	1	ℓ_{64}	ℓ_{65}	1
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
JL:	1	1	2	2	3	2	3	4	2	3	4	5	4	5	6
IL:	1	2	4	6	9	13	16								

U:	1	u_{12}	1	u_{23}	u_{24}	u_{25}	1	u_{34}	u_{35}	1	u_{45}	u_{46}	1	u_{56}	1
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
JU:	1	2	2	3	4	5	3	4	5	4	5	6	5	6	6
IU:	1	3	7	10	13	15	16								

Figure 3.2.

---

they are always equal to 1 and occur as the first stored entry of each row.

Figure 3.3b shows the data structures required when the diagonal entries are omitted. We now note that the indices in JU for certain rows of U are actually final subsequences of the indices for previous rows. For example, the indices for row 3 are 4,5, while those for row 2 are 3,4,5. Instead of storing the indices for row 3 separately, we can simply make use of the last two indices stored for row 2. All that is required is a pointer to locate the indices for row 3.

In general, the indices in JU can be compressed by deleting the indices for any row if they already appear as a final subsequence of some previous row (see Figure 3.4). It is possible to compress the indices in certain other cases as well, but tests have shown that very little is gained by so doing.

Since the compressed indices in JU do not correspond directly to the nonzeroes stored in

$$U = \begin{bmatrix} 1 & u_{12} & 0 & 0 & 0 & 0 \\ 0 & 1 & u_{23} & u_{24} & u_{25} & 0 \\ 0 & 0 & 1 & u_{34} & u_{35} & 0 \\ 0 & 0 & 0 & 1 & u_{45} & u_{46} \\ 0 & 0 & 0 & 0 & 1 & u_{56} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

a

U:	1	u_{12}	1	u_{23}	u_{24}	u_{25}	1	u_{34}	u_{35}	1	u_{45}	u_{46}	1	u_{56}	1
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
JU:	1	2	2	3	4	5	3	4	5	4	5	6	5	6	6
IU:	1	3	7	10	13	15	16								

b

U:	u_{12}	u_{23}	u_{24}	u_{25}	u_{34}	u_{35}	u_{45}	u_{46}	u_{56}
k:	1	2	3	4	5	6	7	8	9
JU:	2	3	4	5	4	5	5	6	6
IU:	1	2	5	7	9	10	10		

c

U:	u_{12}	u_{23}	u_{24}	u_{25}	u_{34}	u_{35}	u_{45}	u_{46}	u_{56}
k:	1	2	3	4	5	6	7	8	9
JU:	2	3	4	5	5	6			
IU:	1	2	5	7	9	10	10		
ISU:	1	2	3	5	6	6			

Figure 3.3.

---

the array U, an extra array of pointers (IJU) is required to locate the indices for each row (see Figure 3.3c). Thus the storage overhead for the compressed storage of U is the number of locations required for IU, JU, and IJU. Although this overhead can be larger than that with the uncompressed scheme, there are important cases in which it is substantially smaller (cf. Section 6; Eisenstat, Schultz, and Sherman [3,4]; Sherman [8]).

Before compaction:

JU:	2	3	4	5	4	5	5	6	6
k:	1	2	3	4	5	6	7	8	9
IU:	1	2	5	7	9	10	10		

After compaction:

JU:	2	3	4	5	5	6	
k:	1	2	3	4	5	6	7
IU:	1	2	5	7	9	10	10
ISU:	1	2	3	5	6	6	

Locations of column indices:	Before compaction	After compaction
row 1	JU(1)	JU(1)
row 2	JU(2) - JU(4)	JU(2) - JU(4)
row 3	JU(5) - JU(6)	JU(3) - JU(4)
row 4	JU(7) - JU(8)	JU(5) - JU(6)
row 5	JU(9)	JU(6)
row 6	-	-

Figure 3.4.

---

4. Three Implementation Designs

In this section we describe three specific implementation designs, which illustrate some of the tradeoffs mentioned earlier. Designs other than these three can also be derived, but these indicate the broad spectrum of implementations that are possible.

The first implementation (SGE1) is designed for speed. It uses the uncompressed storage scheme for A, L, and U because of the smaller operational overhead associated with it. Furthermore, we combine the NUMFAC and SOLVE steps to avoid saving the numeric entries of L, so that the computation consists of a SYMFAC step followed by the NUMSLV step.

The second implementation (SGE2) is designed to reduce the storage requirements. The entire computation is performed in a TRKSLV step to avoid storing either the description or the numerical entries of L. Moreover, U is stored with the compressed storage scheme to reduce the storage overhead. This design incurs more operational overhead than SGE1; the total storage requirements, however, are much smaller.

Finally, the third implementation (SGE3) attempts to balance the design goals of speed and small size. It splits the computation as in SGE1 to avoid storing the numerical entries of L and it uses the compressed storage scheme as in SGE2 to reduce storage overhead.

5. Quantitative Comparisons

In order to compare the designs of Section 4, we tested them on five-point model problems, i.e. linear systems of the form

$$A_n x = b, \tag{5.1}$$

where A_n is a permutation $P_n \tilde{A}_n P_n^T$ of the $n \times n$ block tridiagonal matrix $\tilde{A}$ given by

$$\tilde{A}_n = \begin{bmatrix} B & C & & \\ D & \ddots & \ddots & \\ & \ddots & \ddots & C \\ & & D & B \end{bmatrix}.$$

Here the blocks B, C, and D are $n \times n$ with B tridiagonal and C and D diagonal, and P_n is an $n^2 \times n^2$ permutation matrix chosen to reduce the number of nonzeroes in the factors of A_n and the amount of arithmetic required to solve (5.1). Systems like (5.1) arise frequently in the numerical solution of partial differential equations in rectangular domains (cf. Woo, Eisenstat, Schultz, and Sherman [9] for a specific example.)

Our experiments were designed to answer two questions. First, what are the storage and CPU-time costs of solving the model systems with each of the three implementations? Second, how large a model system can each of the implementations solve in a fixed amount of core storage (262,144 words)?

The results shown in Tables 5.1 and 5.2 were obtained on an IBM System/370 Model 158

n	SGE1	SGE2	SGE3	n	SGE1	SGE2	SGE3
20	17,324	12,789	15,508	60	214,788	137,621	164,364
25	29,038	20,971	25,519	65	257,510	—	194,547
30	44,226	30,909	37,334	70	>262,144	—	227,740
40	85,696	57,653	69,352	75	>262,144	—	>262,144
50	142,382	94,021	113,346	80	>262,144	256,155	>262,144

Table 5.1.
Storage Requirements for the Model Problem.

n	SGE1 (NUMSLV)	SGE2 (TRKSLV)	SGE3 (NUMSLV)	n	SGE1 (NUMSLV)	SGE2 (TRKSLV)	SGE3 (NUMSLV)
20	.58	1.15	.71	60	18.04	31.55	20.31
25	1.14	2.23	1.40	65	--	--	25.73
30	2.05	3.86	2.45	70	--	--	31.86
40	5.00	9.25	5.91	80	--	78.15	--
50	10.22	18.07	11.52				

Table 5.2.
Timings for the Model Problem (in seconds).

using the FORTRAN IV Level H Extended compiler.* For n = 60, SGE1 is the fastest implementation, requiring 40-45% less time than SGE2 and 10-15% less time than SGE3. On the other hand, SGE2 requires less storage, using 35-40% less than SGE1 and 15-20% less than SGE3. Furthermore, we see that SGE2 can solve larger problems (n = 80) than either SGE1 (n = 65) or SGE3 (n = 70). Evidently, then, the qualitative comparisons suggested in Section 4 are borne out in practice.

6. Conclusion

In this paper we have considered the design of implementations of sparse Gaussian elimination in terms of the competing goals of flexibility, speed, and small size. We have seen that by varying certain aspects of the design, it is possible to vary the degree to which each of these goals is attained. Indeed, there seems to be almost a continuous spectrum of possible designs -- SGE1 and SGE2 are its endpoints, while SGE3 is just one of many intermediate points. There is no single implementation that is always best; the particular implementation that should be used in a given situation depends on the problems to be solved and the computational

* We are indebted to Dr. P. T. Woo of the Chevron Oil Field Research Company for running these experiments for us.

environment in which the calculations are to be performed.

References

[1] A. Chang.
Application of sparse matrix methods in electric power system analysis.
In Willoughby, editor, Sparse Matrix Proceedings, 113-122. IBM Research Report RA1, Yorktown Heights, New York, 1968.

[2] A. R. Curtis and J. K. Reid.
The solution of large sparse unsymmetric systems of linear equations.
JIMA 8:344-353, 1971.

[3] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Application of sparse matrix methods to partial differential equations.
Proceedings of the AICA International Symposium on Computer Methods for PDE's, 40-45. Bethlehem, Pennsylvania, 1975.

[4] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Efficient implementation of sparse symmetric Gaussian elimination.
Proceedings of the AICA International Symposium on Computer Methods for PDE's, 33-39. Bethlehem, Pennsylvania, 1975.

[5] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Subroutines for the efficient implementation of sparse Gaussian elimination.
To appear.

[6] F. G. Gustavson.
Basic techniques for solving sparse systems of linear equations.
In Rose and Willoughby, editors, Sparse Matrices and Their Applications, 41-52. Plenum Press, New York, 1972.

[7] W. C. Rheinboldt and C. K. Mesztenyi.
Programs for the solution of large sparse matrix problems based on the arc-graph structure.
Technical Report TR-262, Computer Science Center, University of Maryland, 1973.

[8] A. H. Sherman.
On the Efficient Solution of Sparse Systems of Linear and Nonlinear Equations.
PhD dissertation, Yale University, 1975.

[9] P. T. Woo, S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Application of sparse matrix techniques to reservoir simulation.
Proceedings of the Symposium on Sparse Matrix Computations, Argonne National Laboratories, 1975.